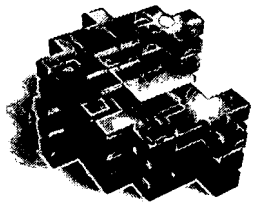
APPENDIX A

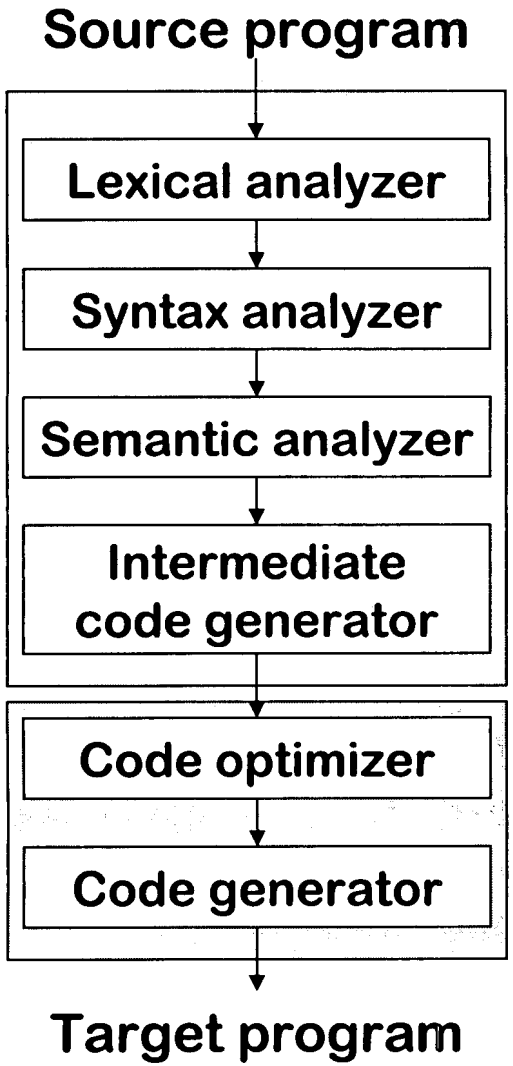
Lexical Analysis & Syntactic Analysis

CS 671

January 24, 2008

Last Time

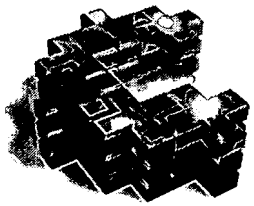




Lexical Analyzer

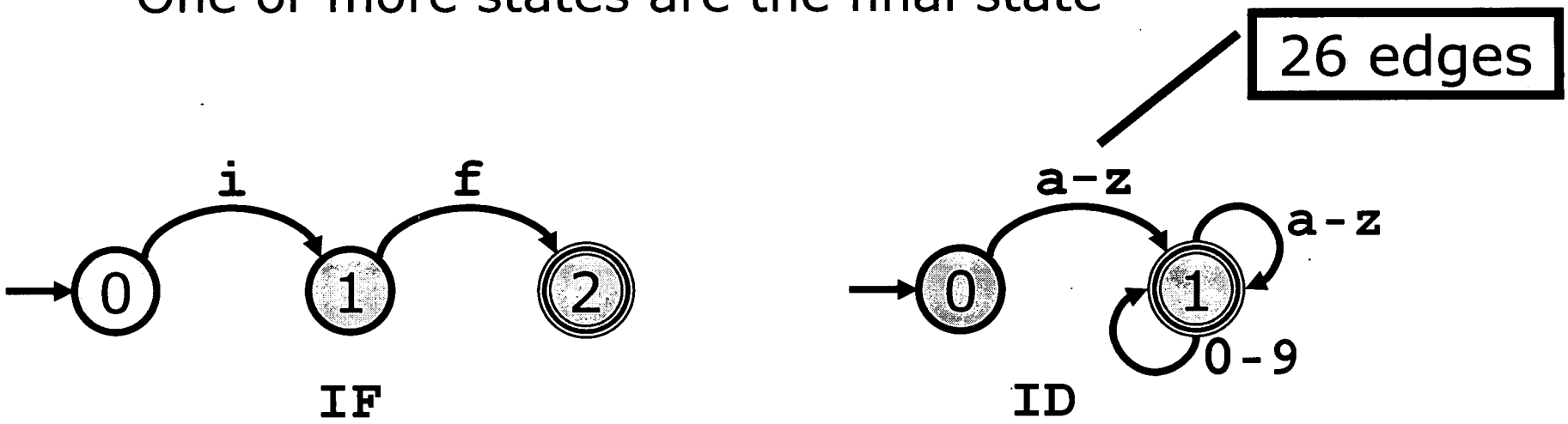
- Group sequence of characters into lexemes – smallest meaningful entity in a language (keywords, identifiers, constants)
- Characters read from a file are buffered – helps decrease latency due to i/o. Lexical analyzer manages the buffer
- Makes use of the theory of regular languages and finite state machines
- Lex and Flex are tools that construct lexical analyzers from regular expression specifications

Finite Automata

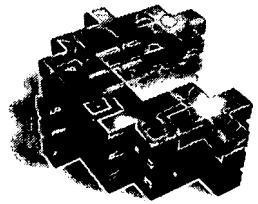


Takes an input string and determines whether it's a valid sentence of a language

- A finite automaton has a finite set of states
- Edges lead from one state to another
- Edges are labeled with a symbol
- One state is the start state
- One or more states are the final state



Finite Automata



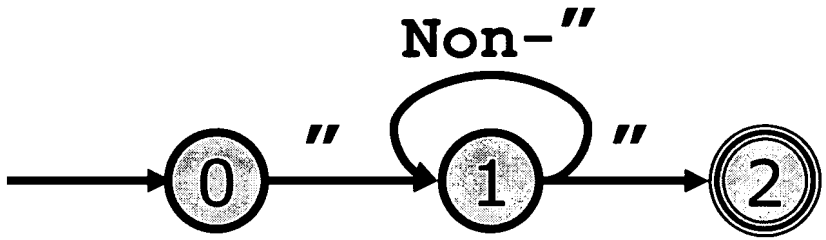
Automaton (DFA) can be represented as:

- A transition table

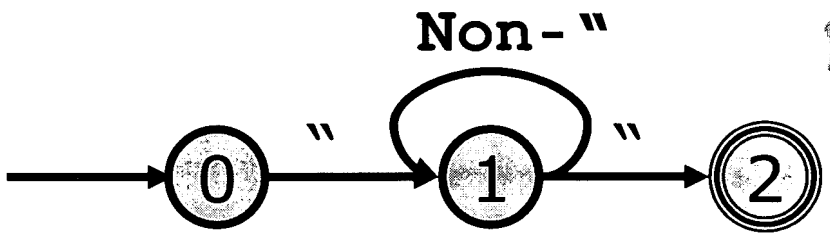
\" [^"]* \"

	"	non-"
0		
1		
2		

- A graph

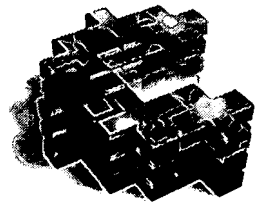


Implementation



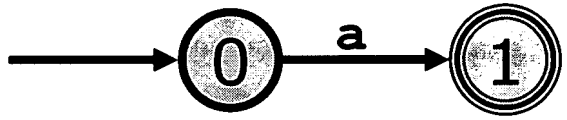

```
boolean accept_state[NSTATES] = { … };
int trans_table[NSTATES][NCHARS] = { … };
int state = 0;

while (state != ERROR_STATE) {
      c = input.read();
      if (c < 0) break;
      state = table[state][c];
}
return accept_state[state];
```

RegExp → Finite Automaton

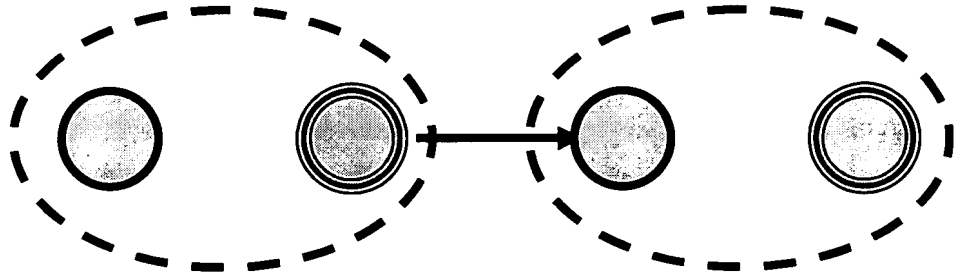


- Can we build a finite automaton for every regular expression?
- Strategy: consider every possible kind of regular expression (define by induction)

a

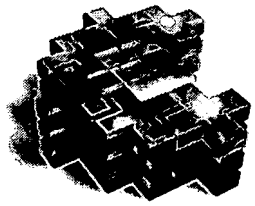


R_1R_2



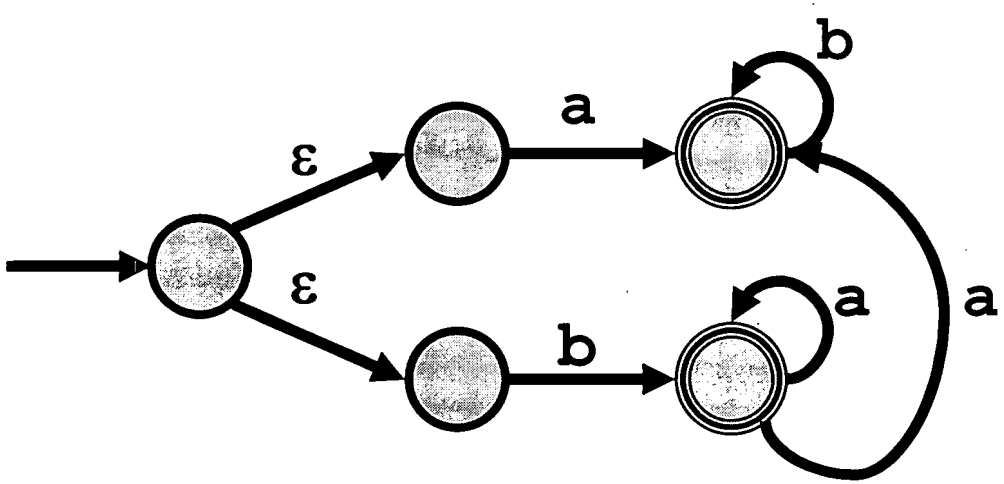
$R_1 \mid R_2$?

Deterministic vs. Nondeterministic

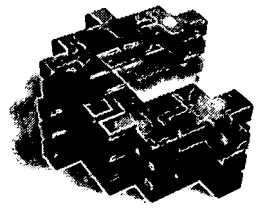


Deterministic finite automata (DFA) – No two edges from the same state are labeled with the same symbol

Nondeterministic finite automata (NFA) – may have arrows labeled with ε (which does not consume input)

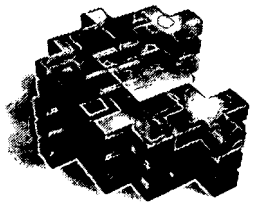


DFA vs. NFA

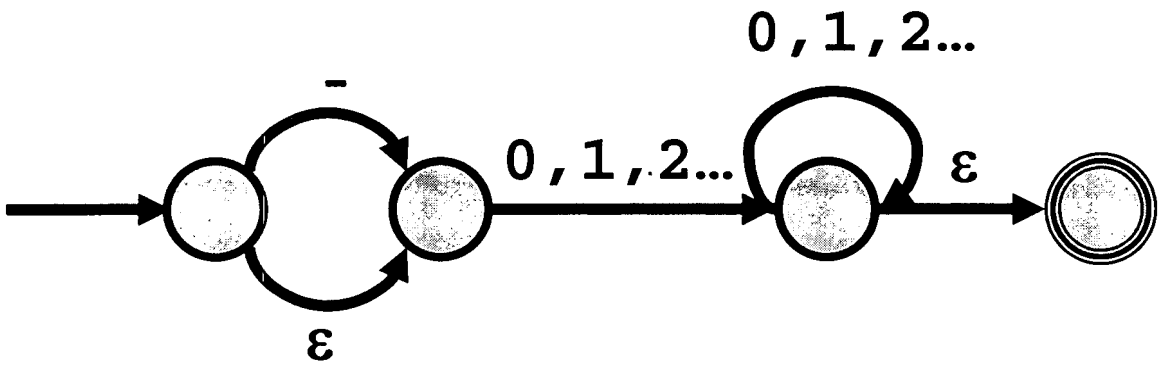


- **DFA**: action of automaton on each input symbol is fully determined
  - obvious table-driven implementation
- **NFA**:
  - automaton may have choice on each step
  - automaton accepts a string if there is *any way* to make choices to arrive at accepting state / every path from start state to an accept state is a string accepted by automaton
  - not obvious how to implement efficiently!

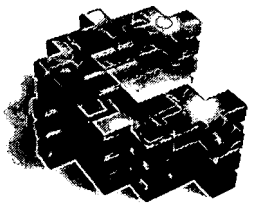
RegExp → NFA

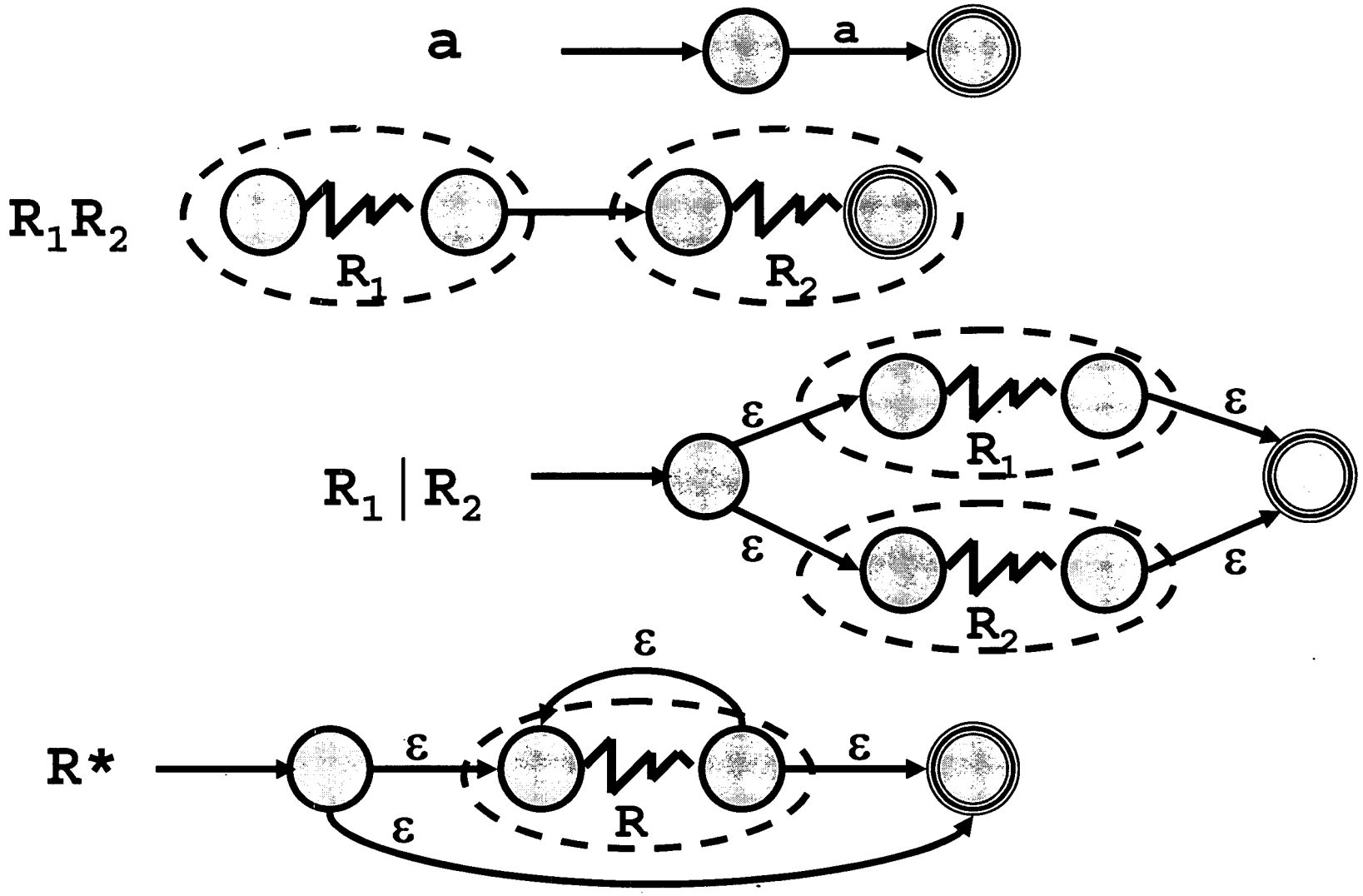


-? [0-9]+ **(-|ε) [0-9][0-9]***

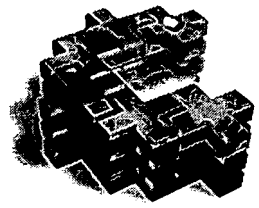


Inductive Construction





Executing NFA

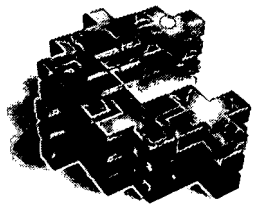


- **Problem**: how to execute NFA efficiently?

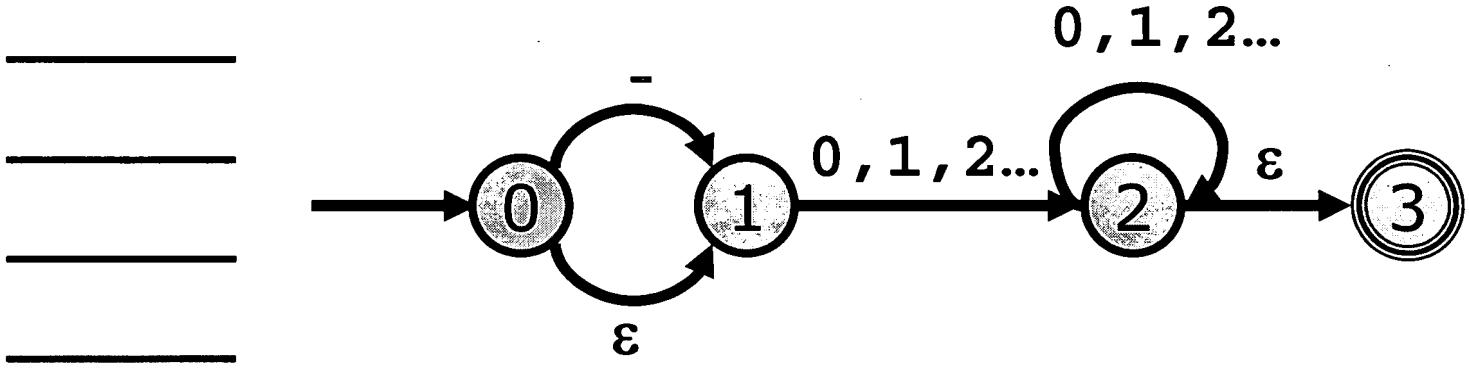
"strings accepted are those for which there is some corresponding path from start state to an accept state"

- **Conclusion**: search all paths in graph consistent with the string
- **Idea**: search paths in parallel
  - Keep track of subset of NFA states that search could be in after seeing string prefix
  - "Multiple fingers" pointing to graph

Example

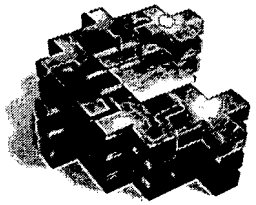


- **Input string: -23**
- **NFA States**

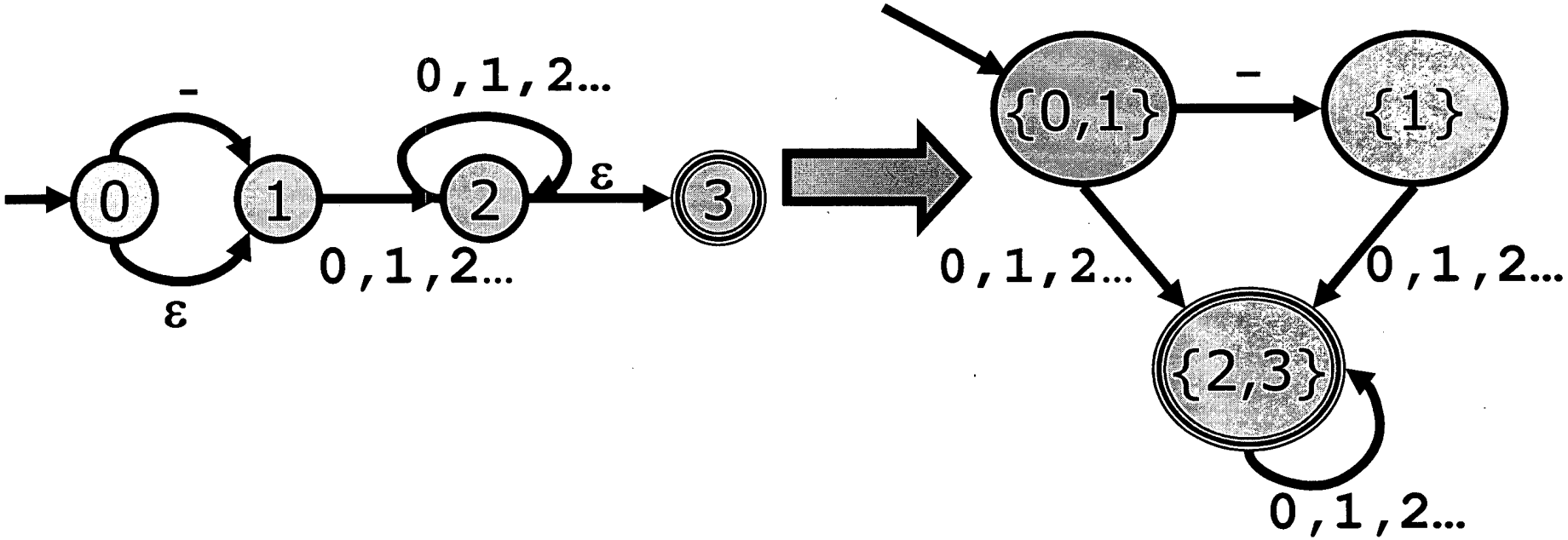


- **Terminology: *ε-closure* - set of all reachable states without consuming any input**
  - *ε-closure* of 0 is {0,1}

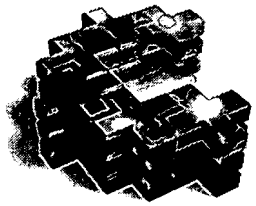
NFA→DFA Conversion



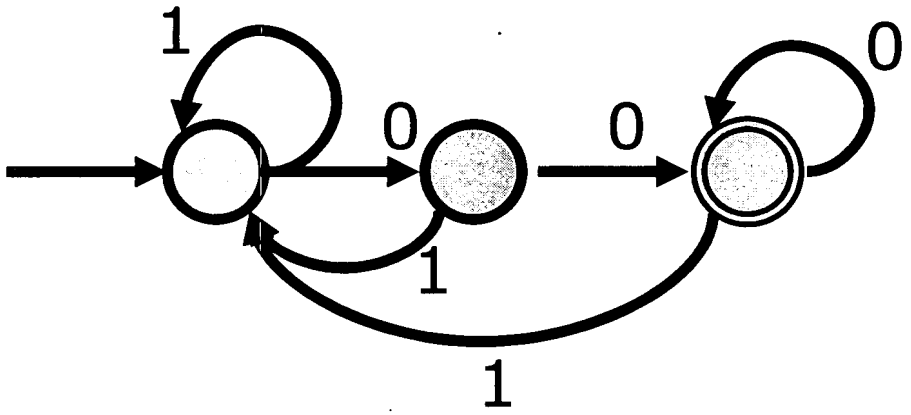
- Can convert NFA directly to DFA by same approach
- Create one DFA for each distinct subset of NFA states that could arise
- States: {0,1}, {1}, {**2, 3**}



DFA Minimization



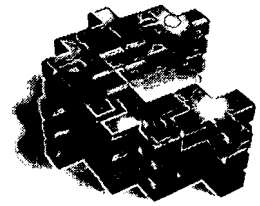
- DFA construction can produce large DFA with many states
- Lexer generators perform additional phase of *DFA minimization* to reduce to minimum possible size



What does this DFA do?

Can it be simplified?

Automatic Scanner Construction



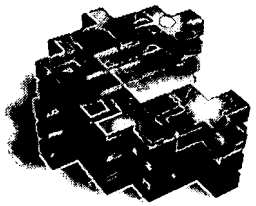
To convert a specification into code

1. Write down the RE for the input language
2. Build a big NFA
3. Build the DFA that simulates the NFA
4. Systematically shrink the DFA
5. Turn it into code

Scanner generators

- Lex and flex work along these lines
- Algorithms are well known and understood
- Key issue is interface to the parser

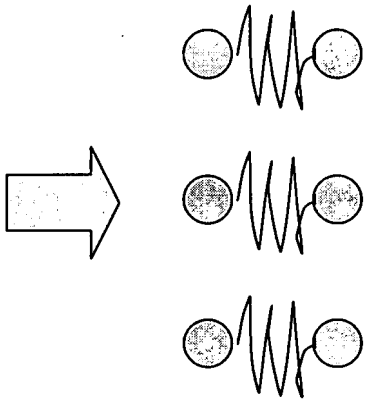
Building a Lexer



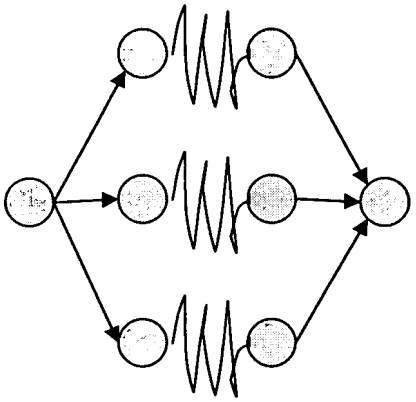
Specification

```
"if"
"while"
[a-zA-Z][a-zA-Z0-9]*
[0-9][0-9]*
(
)
```

NFA for each RE



Giant NFA



Giant DFA

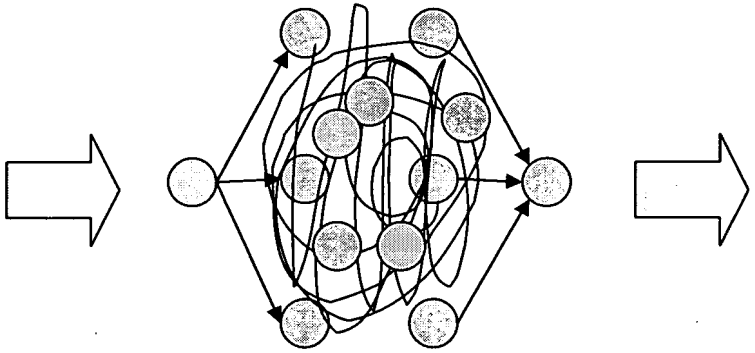
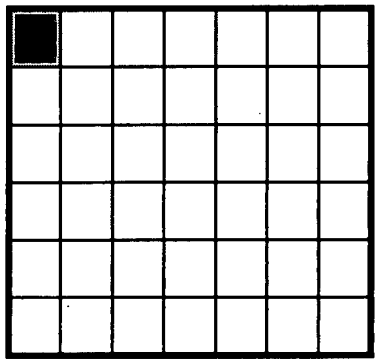
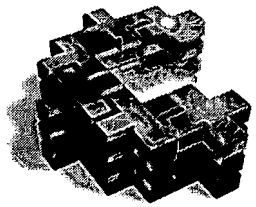


Table-driven code

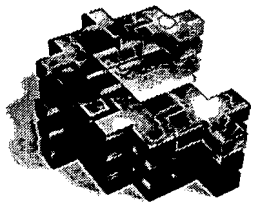


Lexical Analysis Summary



- **Regular expressions**
  - –efficient way to represent languages
  - –used by lexer generators
- **Finite automata**
  - –describe t he actual implementation of a lexer
- **Process**
  - –Regular exp ressions (+priority) converted to NFA
  - –NFA converted to DFA

Where Are We?

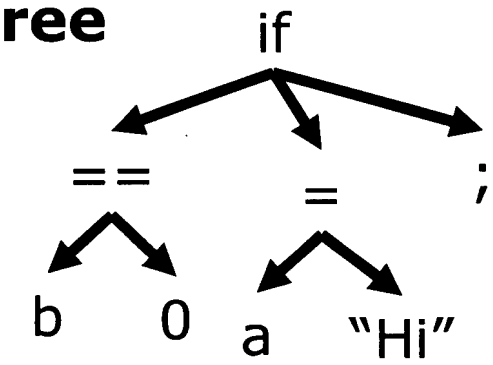


Source code: `if (b==0) a = "Hi";`

Token Stream:

if	(	b	==	0	)	a	=	"Hi"	;

Abstract Syntax Tree (AST)



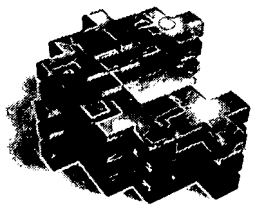
Lexical Analysis

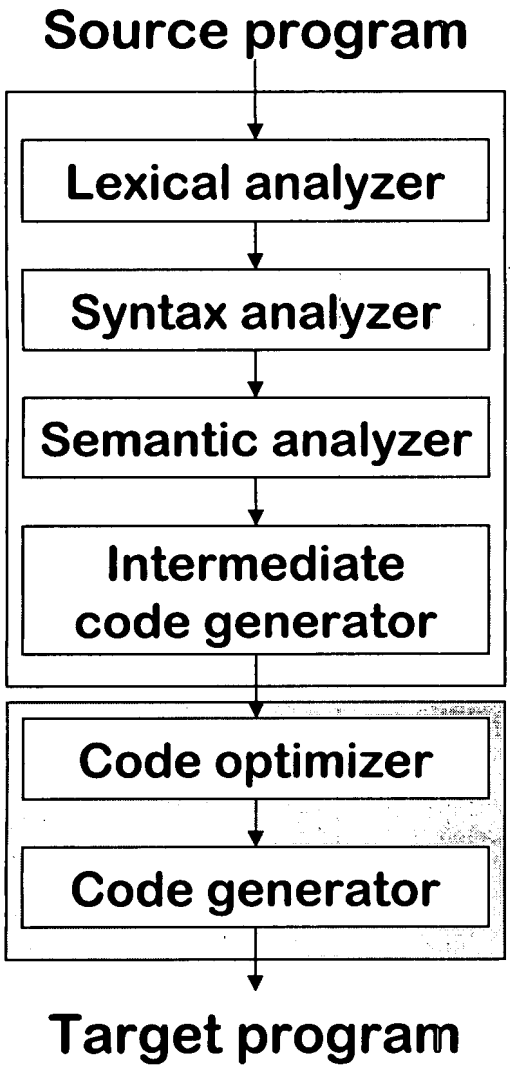
Syntactic Analysis

Semantic Analysis

Do tokens conform to the language syntax?

Phases of a Compiler

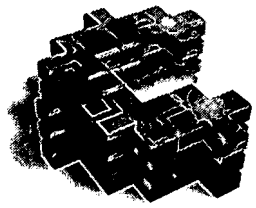




Parser

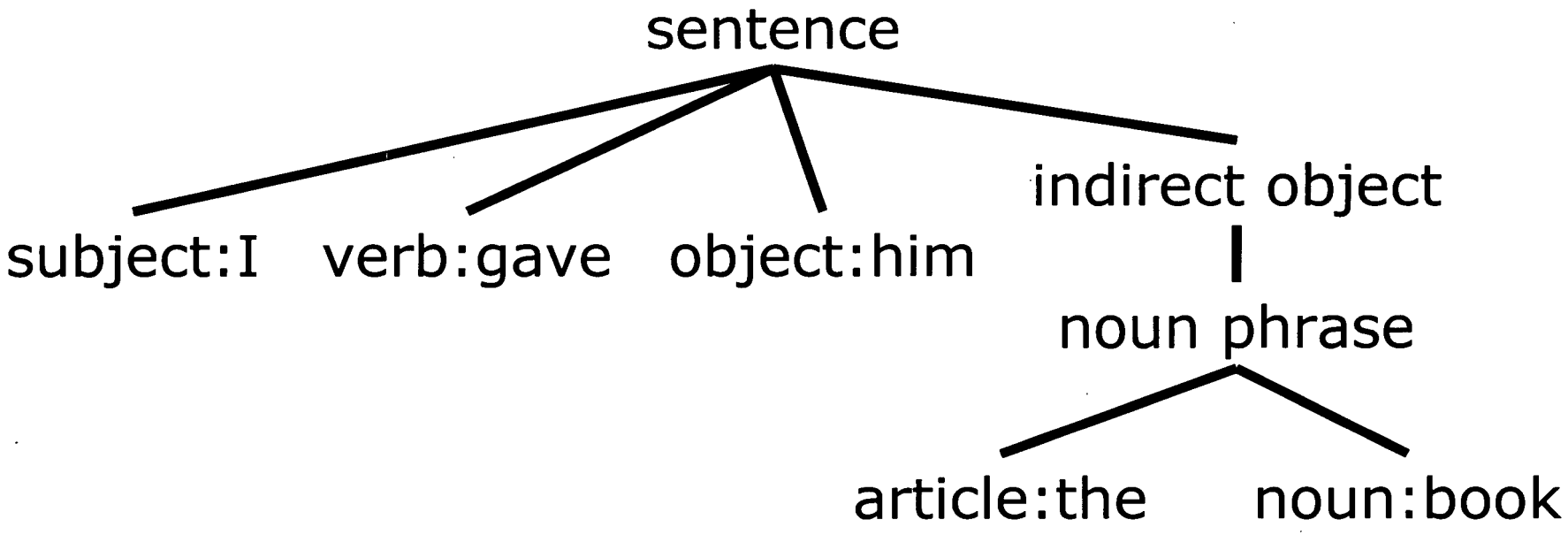
- Convert a linear structure – sequence of tokens – to a hierarchical tree-like structure – an AST
- The parser imposes the syntax rules of the language
- Work should be linear in the size of the input (else unusable) → type consistency cannot be checked in this phase
- Deterministic context free languages and pushdown automata for the basis
- Bison and yacc allow a user to construct parsers from CFG specifications

What is Parsing?

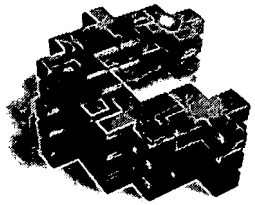


Parsing: Recognizing whether a sentence (or program) is grammatically well formed and identifying the function of each component

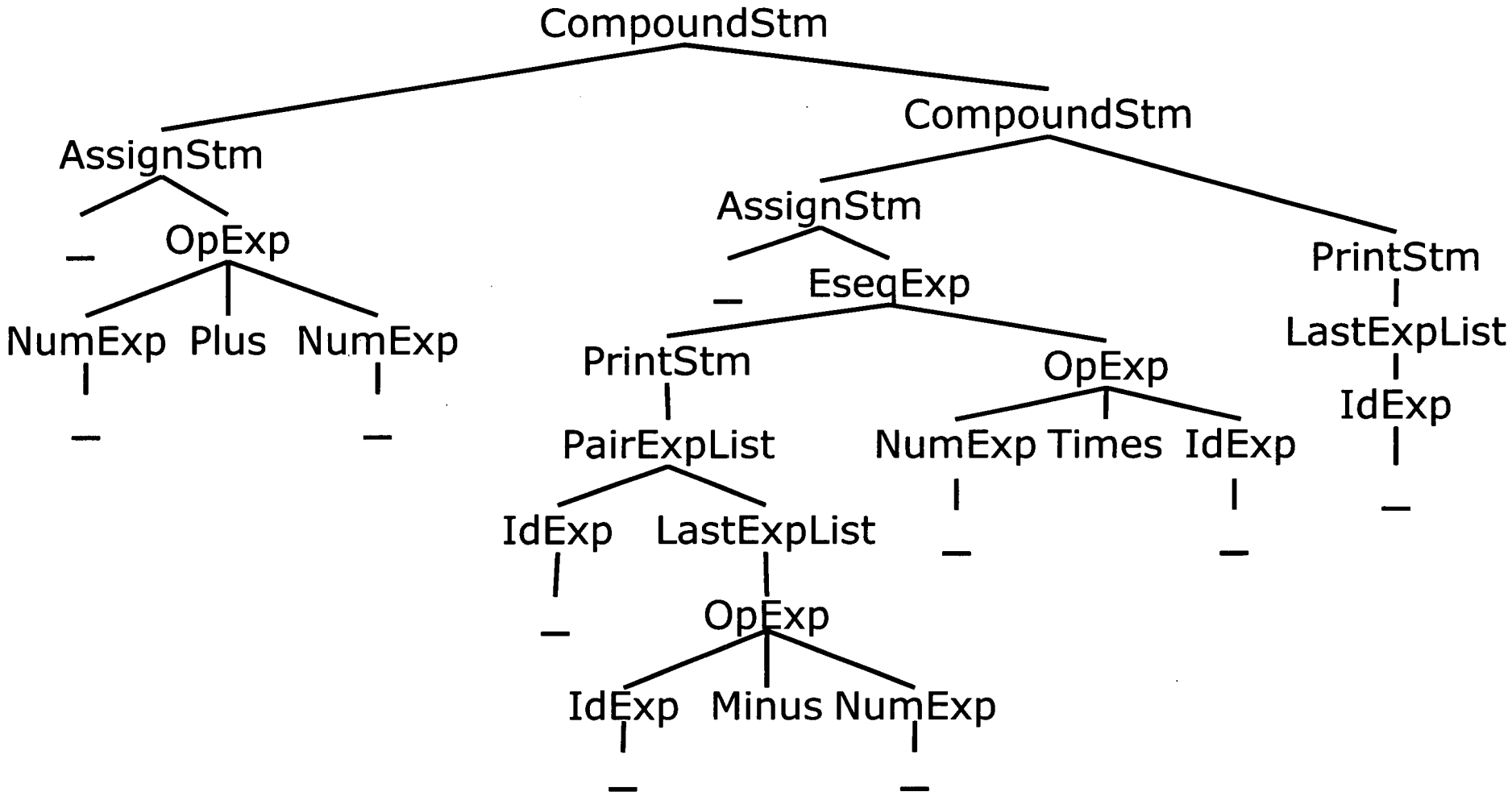
"I gave him the book"



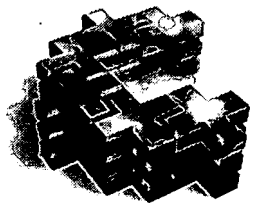
Tree Representations



a = 5+3 ; b = (print (a , a–1) , 10*a); print(b)

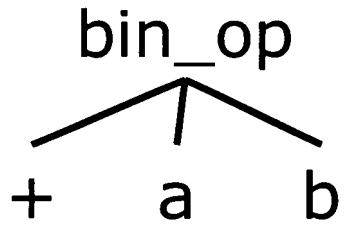


Overview of Syntactic Analysis

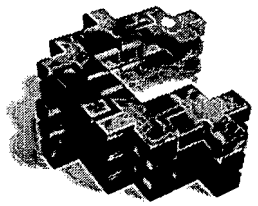


- **Input**: stream of tokens
- **Output**: abstract syntax tree
- **Implementation**:
  - Parse token stream to traverse concrete syntax (parse tree)
  - During traversal, build abstract syntax tree
  - Abstract syntax tree removes extra syntax

$$a + b \approx (a) + (b) \approx ((a)+((b)))$$



What Parsing Doesn't Do

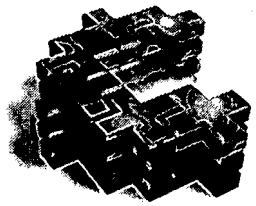


- **Doesn't check:** type agreement, variable declaration, variable initialization, etc.

```
int x = true;
int y;
z = f(y);
```

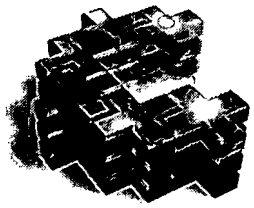
- **Deferred** until semantic analysis

Specifying Language Syntax

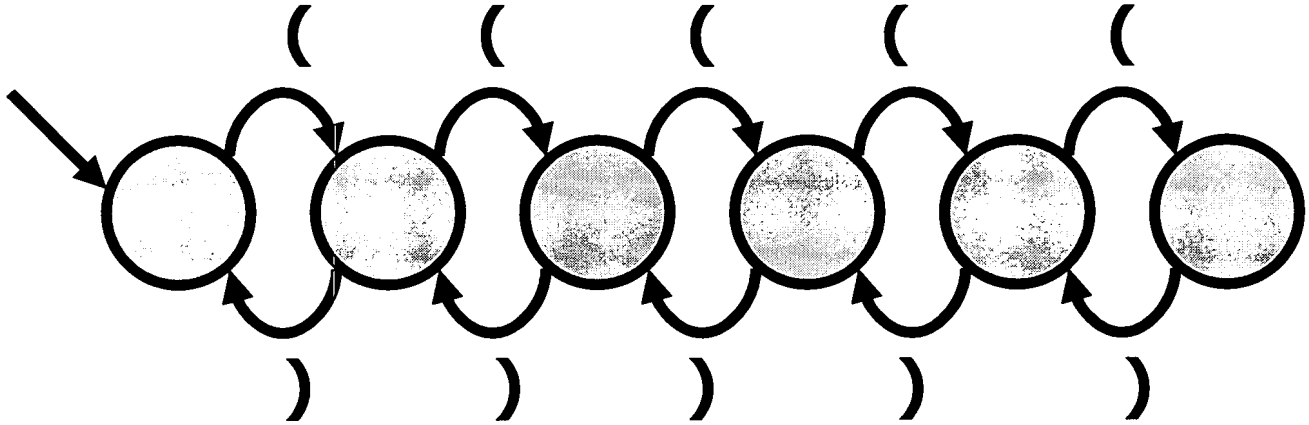


- **First problem:** how to describe language syntax precisely and conveniently
- **Last time:** can describe tokens using regular expressions
- Regular expressions easy to implement, efficient (by converting to DFA)
- Why not use regular expressions (on tokens) to specify programming language syntax?

Need a More Powerful Representation

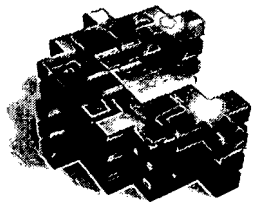


- Programming languages are **not regular**
  - –can not be described by regular expressions
- **Consider:** language of all strings that contain balanced parentheses



- DFA has only finite number of states
- Cannot perform unbounded counting

Context-Free Grammars



- A specification of the balanced-parenthesis language:

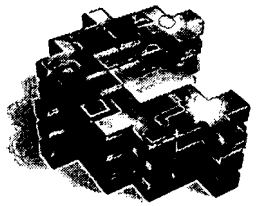
$$S \rightarrow (S) S$$

$$S \rightarrow \varepsilon$$

- The definition is recursive
- A **context-free grammar**
  - More expressive than regular expressions
  - $S = (S)\ \varepsilon = ((S)\ S)\ \varepsilon = ((\varepsilon)\ \varepsilon)\ \varepsilon = (())$

If a grammar accepts a string, there is a **derivation** of that string using the productions of the grammar

Context-Free Grammar Terminology



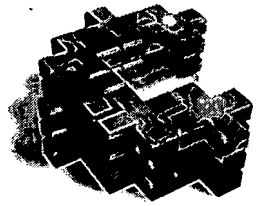
- **Terminals**
  - Token or ε
- **Non-terminals**
  - Syntactic variables

$$S \rightarrow (S)\ S$$

$$S \rightarrow \varepsilon$$

- **Start symbol**
  - A special nonterminal is designated (*S*)
- **Productions**
  - Specify how non-terminals may be expanded to form strings
  - LHS: single non-terminal, RHS: string of terminals or non-terminals
- Vertical bar is shorthand for multiple productions

Sum Grammar



$S \rightarrow E + S \mid E$

$E \rightarrow$ **number** $\mid (S)$

e.g. (1 + 2 + (3+4))+5

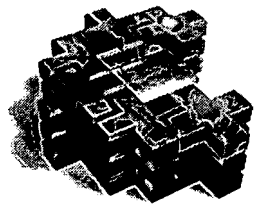
$S \rightarrow E + S$

$S \rightarrow E$

$E \rightarrow$ number

$E \rightarrow (S)$

_ productions
_ non-terminals:
_ terminals:
start symbol S

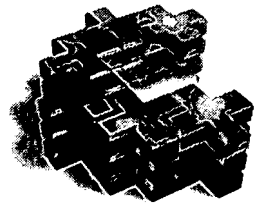


Develop a Context-Free Grammar for ...

1. $a^n b^n c^n$

2. $a^m b^n c^{m+n}$

Constructing a Derivation



- Start from start symbol (*S*)
- Productions are used to derive a sequence of tokens from the start symbol
- For arbitrary strings α, β and γ and a production $A \rightarrow \beta$

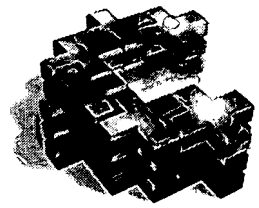
A single step of derivation is $\alpha A \gamma \Rightarrow \alpha \beta \gamma$

–*i.e.*, substitute β for an occurrence of A

(**S** + E) + E → (**E + S** + E) + E

(A = S, β = E + S)

Derivation Example



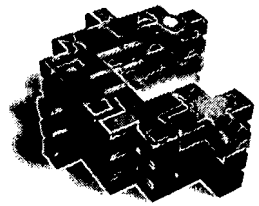
$$S \rightarrow E + S \mid E$$

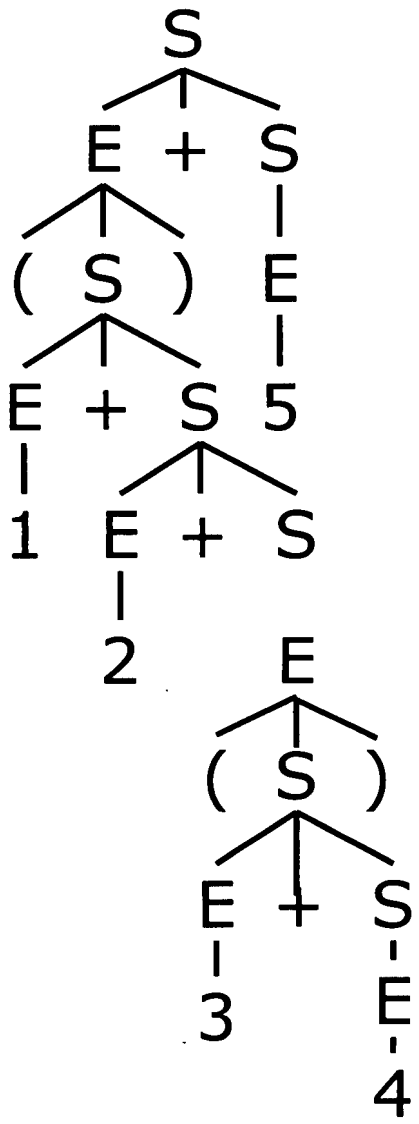
$$E \rightarrow \text{number} \mid (S)$$

Derive (1+2+ (3+4))+5:

$S \rightarrow E + S \rightarrow$

Derivation ⇒ Parse Tree



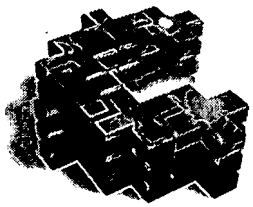


- Tree representation of the derivation
- Leaves of tree are terminals; in-order traversal yields string
- Internal nodes: non-terminals
- No information about order of derivation steps

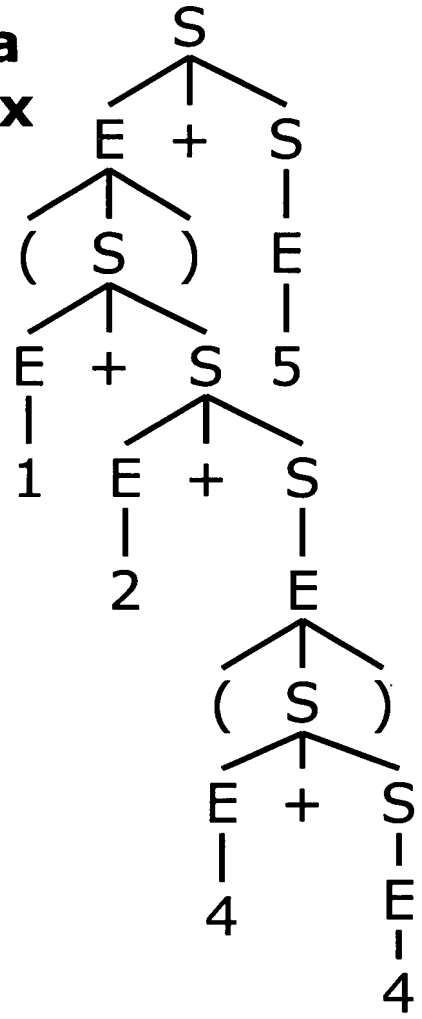
⇐ (1+2+ (3+4))+5

$S \rightarrow E + S \mid E$
$E \rightarrow$ number $\mid$ (S)

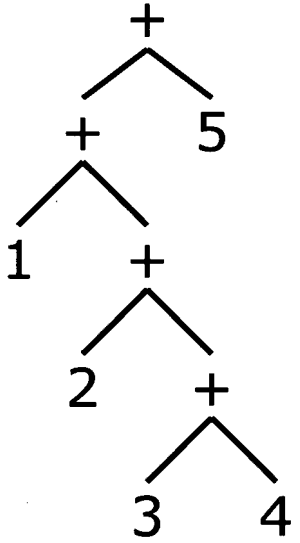
Parse Tree vs. AST



Parse Tree, aka concrete syntax

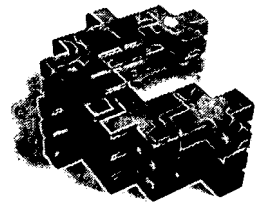


Abstract Syntax Tree



Discards/abstracts unneeded information

Derivation Order



Can choose to apply productions in any order; select any non-terminal A

$$\alpha A \gamma \Rightarrow \alpha \beta \gamma$$

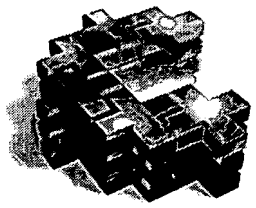
- Two standard orders: left- and right-most -- useful for different kinds of automatic parsing
- **Leftmost derivation**: In the string, find the left-most non-terminal and apply a production to it

$$E + S \rightarrow 1 + S$$

- **Rightmost derivation**: Always choose rightmost non-terminal

$$E + S \rightarrow E + E + S$$

Example



$S \rightarrow E + S \mid E$

$E \rightarrow \text{number} \mid (S)$

Left-Most Derivation

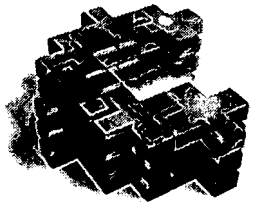
$S \rightarrow E+S \rightarrow (S) + S \rightarrow (E + S)+ S \rightarrow (1 + S)+S \rightarrow$
$(1+E+S)+S \rightarrow (1+2+S)+S \rightarrow (1+2+E)+S \rightarrow (1+2+(S))+S$
$\rightarrow (1+2+(E+S))+S \rightarrow (1+2+(3+S))+S \rightarrow (1+2+(3+E))+S$
$\rightarrow (1+2+(3+4))+S \rightarrow (1+2+(3+4))+E \rightarrow (1+2+(3+4))+5$

Right-Most Derivation

$S \rightarrow E+S \rightarrow E+E \rightarrow E+5 \rightarrow (S)+5 \rightarrow (E+S)+5 \rightarrow (E+E+S)+5 \rightarrow$
$(E+E+E)+5 \rightarrow (E+E+(S))+5 \rightarrow (E+E+(E+S))+5 \rightarrow$
$(E+E+(E+E))+5 \rightarrow (E+E+(E+4))+5 \rightarrow (E+E+(3+4))+5 \rightarrow$
$(E+2+(3+4))+5 \rightarrow (1+2+(3+4))+5$

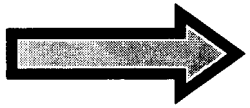
- Same parse tree: same productions chosen, different order

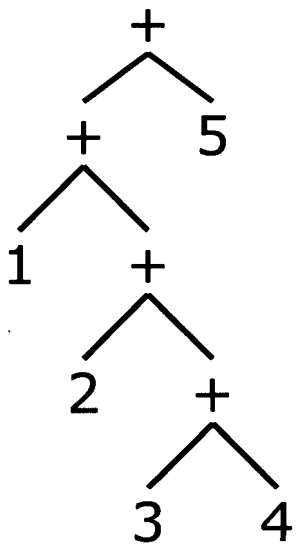
Associativity



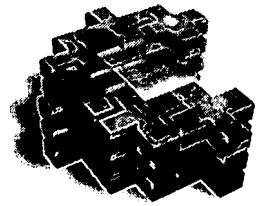
- In example grammar, left-most and right-most derivations produced identical parse trees
- + operator associates to right in parse tree regardless of derivation order

(1+2+(3+4))+5





Another Example

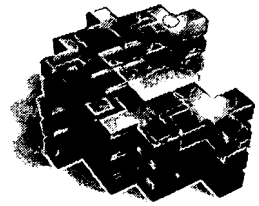


Let's derive the string `x - 2 * y`

#	*Production rule*		
1	*expr*	→	*expr op expr*
2		|	number
3		|	identifier
4	*op*	→	+
5		|	-
6		|	*
7		|	/

Rule	*Sentential form*
-	*expr*
1	*expr op expr*
3	*<id,x> op expr*
5	*<id,x> - expr*
1	*<id,x> - expr op expr*
2	*<id,x> - <num,2> op expr*
6	*<id,x> - <num,2> * expr*
3	*<id,x> - <num,2> * <id,y>*

Left vs. Right derivations



Two derivations of `x - 2 * y`

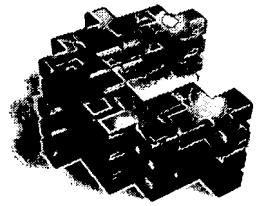
Rule	Sentential form
-	*expr*
1	*expr op expr*
3	*<id, x> op expr*
5	*<id,x> - expr*
1	*<id,x> - expr op expr*
2	*<id,x> - <num,2> op expr*
6	*<id,x> - <num,2> * expr*
3	*<id,x> - <num,2> * <id,y>*

Left-most derivation

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	*expr op <id,y>*
6	*expr * <id,y>*
1	*expr op expr * <id,y>*
2	*expr op <num,2> * <id,y>*
5	*expr - <num,2> * <id,y>*
3	*<id,x> - <num,2> * <id,y>*

Right-most derivation

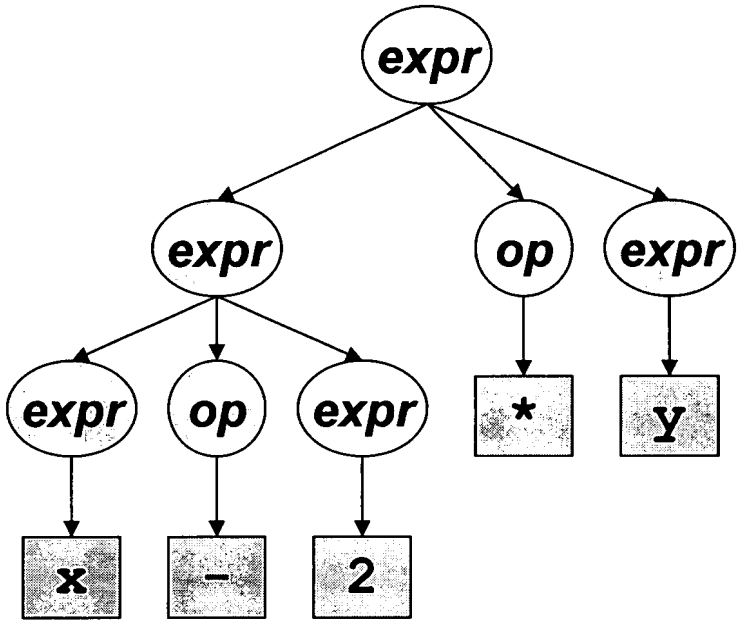
Right-Most Derivation



Right-most derivation

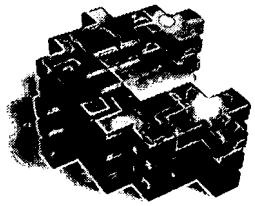
Rule	*Sentential form*
-	*expr*
1	*expr op expr*
3	*expr op <id,y>*
6	*expr * <id,y>*
1	*expr op expr * <id,y>*
2	*expr op <num,2> * <id,y>*
5	*expr - <num,2> * <id,y>*
3	*<id,x> - <num,2> * <id,y>*

Parse tree



Problem: evaluates as `(x - 2) * y`

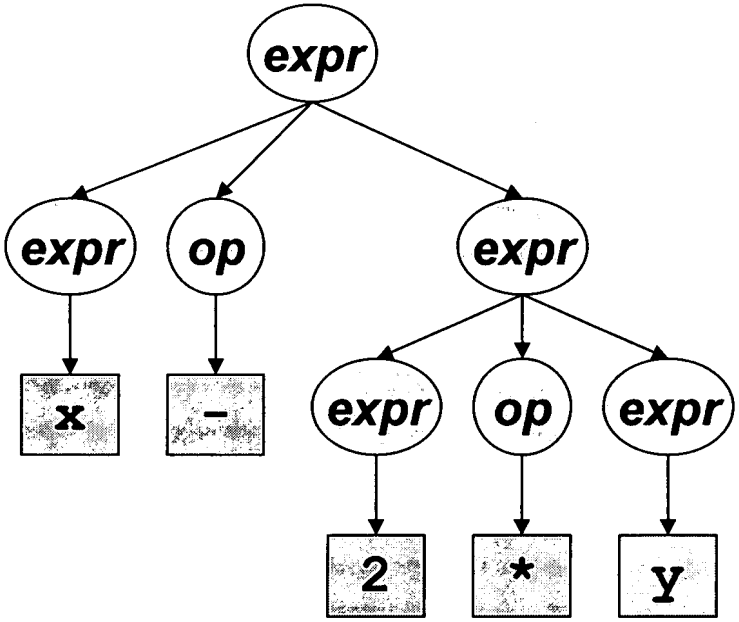
Left-Most Derivation



Left-most derivation

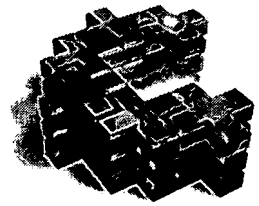
Rule	*Sentential form*
-	*expr*
1	*expr op expr*
3	*<id, x> op expr*
5	*<id,x> - expr*
1	*<id,x> - expr op expr*
2	*<id,x> - <num,2> op expr*
6	*<id,x> - <num,2> * expr*
3	*<id,x> - <num,2> * <id,y>*

Parse tree

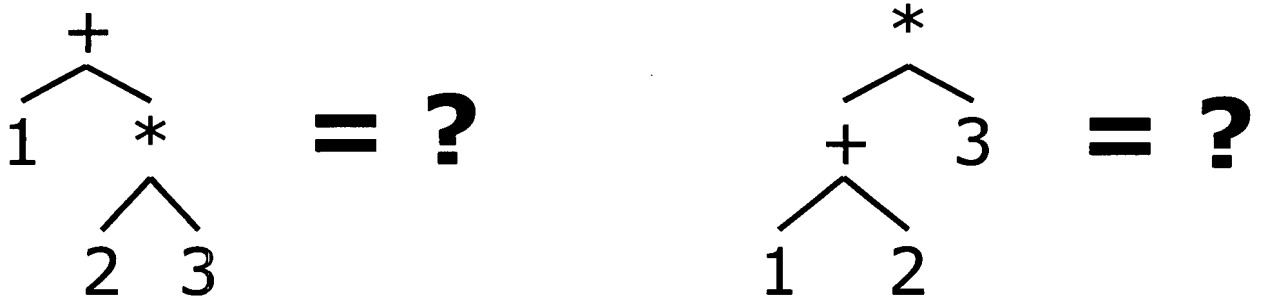


<u>Solution</u>: evaluates as `x - (2 * y)`

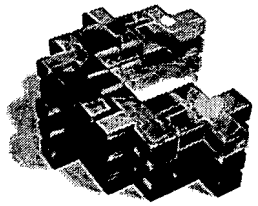
Impact of Ambiguity



- Different parse trees correspond to different evaluations!
- Meaning of program not defined



Derivations and Precedence



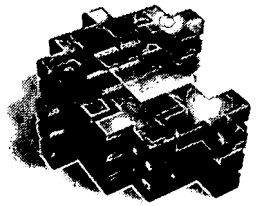
Problem:

- Two different valid derivations
- Shape of tree implies its meaning
- One captures semantics we want – ***precedence***

Can we express precedence in grammar?

- Notice: operations deeper in tree evaluated first
- **Idea**: add an intermediate production
  - New production isolates different levels of precedence
  - Force higher precedence "deeper" in the grammar

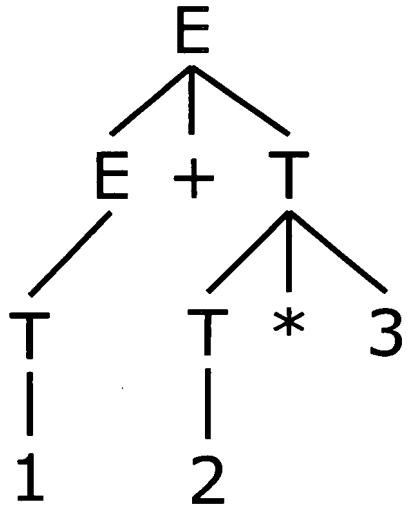
Eliminating Ambiguity



• Often can eliminate ambiguity by adding non-terminals & allowing recursion only on right or left

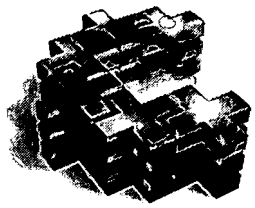
$$Exp \rightarrow Exp + Term \mid Term$$

$$Term \rightarrow Term * \textbf{num} \mid \textbf{num}$$



- New *Term* enforces precedence
- Left-recursion : left-associativity

Adding precedence



A complete view:

Level 1: lower precedence – higher in the tree

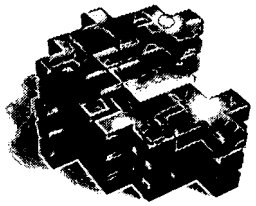
Level 2: higher precedence – deeper in the tree

#	Production rule
1	*expr* → *expr* + *term*
2	| *expr* - *term*
3	| *term*
4	*term* → *term* * *factor*
5	| *term* / *factor*
6	| *factor*
7	*factor* → number
8	| identifier

Observations:

- Larger: requires more rewriting to reach terminals
- Produces same parse tree under both left and right derivations

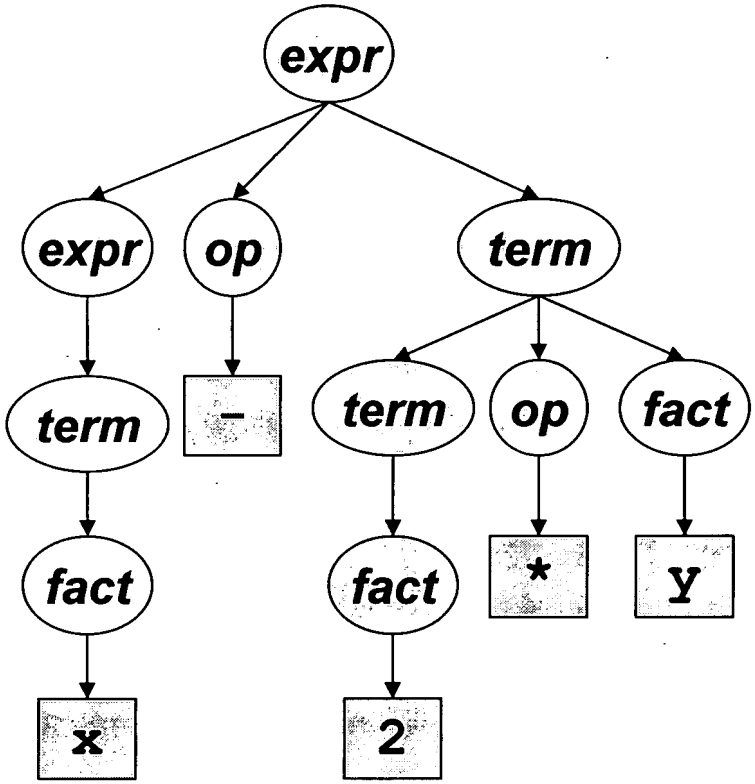
Expression example



Right-most derivation

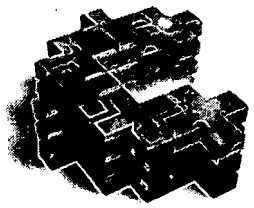
Rule	*Sentential form*
-	*expr*
2	*expr - term*
4	*expr - term * factor*
8	*expr - term * <id,y>*
6	*expr - factor * <id,y>*
7	*expr - <num,2> * <id,y>*
3	*term - <num,2> * <id,y>*
6	*factor - <num,2> * <id,y>*
8	*<id,x> - <num,2> * <id,y>*

Parse tree



⇨ Now right derivation yields `x - (2 * y)`

Ambiguous grammars

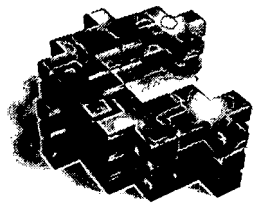


A grammar is ambiguous *iff*:

- There are multiple leftmost or multiple rightmost derivations for a single sentential form
- ***Note***: leftmost and rightmost derivations may differ, even in an unambiguous grammar
- Intuitively:
  - We can choose different non-terminals to expand
  - But each non-terminal should lead to a unique set of terminal symbols

Classic example: if-then-else ambiguity

If-then-else



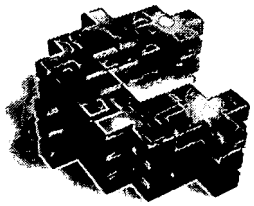
Grammar:

#	*Production rule*
1	*stmt* → `if` *expr* `then` *stmt*
2	| `if` *expr* `then` *stmt* `else` *stmt*
3	| *...other statements...*

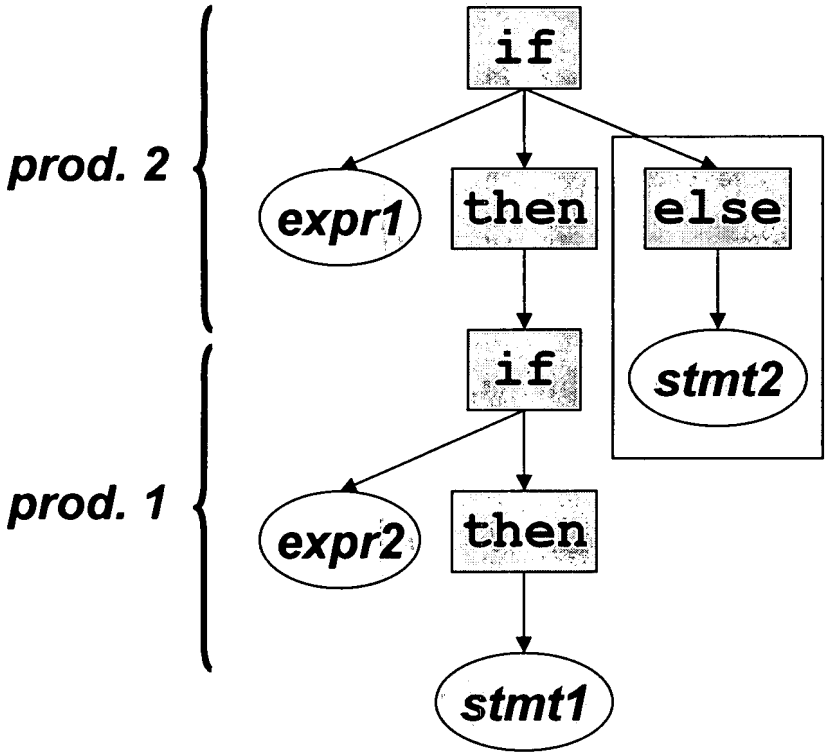
Problem: nested if-then-else statements

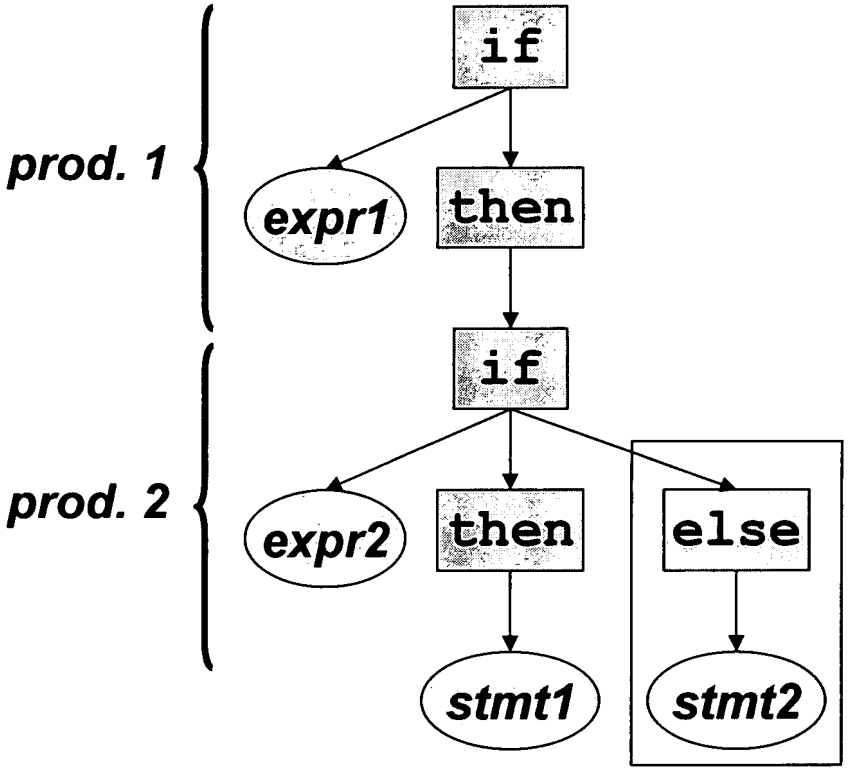
- Each one may or may not have `else`
- How to match each `else` with `if`

If-then-else Ambiguity

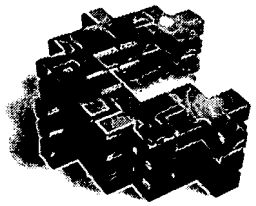


`if` ***expr1*** `then if` ***expr2*** `then` ***stmt1*** `else` ***stmt2***





Removing Ambiguity



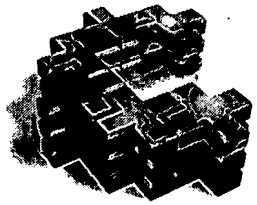
Restrict the grammar

- Choose a rule: "else" matches innermost "if"
- Codify with new productions

#	*Production rule*
1	*stmt* → `if` *expr* `then` *stmt*
2	| `if` *expr* `then` *withelse* `else` *stmt*
3	| *...other statements...*
4	*withelse* → `if` *expr* `then` *withelse* `else` *withelse*
5	| *...other statements...*

- Intuition: when we have an "else", all preceding nested conditions must have an "else"

Limits of CFGs



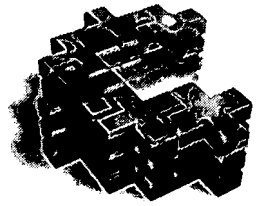
- **Syntactic analysis can't catch all "syntactic" errors**
- **Example: C++**

```
HashTable<Key,Value> x;
```

- **Example: Fortran**

```
x = f(y);
```


Big Picture



Scanners

- Based on regular expressions
- Efficient for recognizing token types
- Remove comments, white space
- Cannot handle complex structure

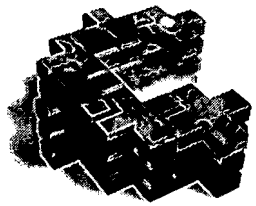
Parsers

- Based on context-free grammars
- More powerful than REs, but still have limitations
- Less efficient

Type and semantic analysis

- Based on attribute grammars and type systems
- Handles "context-sensitive" constructs

Roadmap



So far...

- Context-free grammars, precedence, ambiguity
- Derivation of strings

Parsing:

- Start with string, discover the derivation
- Two major approaches
  - Top-down – start at the top, work towards terminals
  - Bottom-up – start at terminals, assemble into tree

APPENDIX B

Web Development

March 01, 2001

Safe Session Tracking

Choosing the right method is crucial for e-commerce success.

Stan Kim

Session tracking is essential for Web applications, particularly for important e-commerce applications like shopping carts where a user's screen choices must be remembered.What follows is a brief overview of session tracking mechanisms and a recommendation for how to employ them; however, I don't discuss how you might save data that is to be used across multiple sessions that can be handled by servlets writing to a database via an Objective Relational Mapping Framework (ORMF), or directly through Java Database Connectivity (JDBC) or through entity Enterprise Java Beans (EJB). This article also focuses primarily on session tracking using Java servlets. Also, note that session tracking is only an issue for Web browser clients as standalone clients can easily keep track of state locally.

Web applications often need to retain the state of user sessions. For example, when a customer buys books from an online bookseller, he uses an online shopping cart to select books before checking out. The shopping cart is only valid for the session and is abandoned if the shopper leaves the site for a certain amount of time without proceeding to checkout. A user's identity is also session information—because HTTP is connectionless, a server would need to ask the user to identify himself every time he requests a page if some type of session-tracking mechanism wasn't available.

Fortunately, there are mechanisms for saving session information: user authorization, cookies, URL rewriting, hidden form fields, HTTPSession and stateful session Enterprise Java Beans.

Most Web servers allow files or directories to be protected by forcing user authorization. *User authorization* prompts the user for a login and password and then validates the information. Once logged in, a servlet can obtain the remote user's identity by calling the `getRemoteUser()` method of `HTTPServletRequest`. The remote user name can be used to identify the user and session-specific information stored in a shared Java class or in an external database. This technique is easy to implement and allows the user to easily access the Web site from different machines. This technique isn't free of disadvantages, however: The user must enter a login and password to perform session tracking, and using this technique for a shopping site forces the user to create an account and log in prior to adding items to his shopping cart—usually such e-commerce sites allow users to add items to a shopping cart before creating an account.

A *cookie* is data stored with a **Web browser** such as Netscape Navigator or Microsoft Internet Explorer that can be used to perform session tracking. A user's cookie can be created, read or modified server-side using CGI scripts or via servlets. When the Web browser receives a cookie, it's saved to the user's hard disk and subsequently transmitted each time a page is accessed on the server. A servlet can read a user's cookies using method `getCookies()` of `HTTPServletRequest`, and a servlet can create and set a cookie using `addCookie(Cookie cookie)` of `HTTPServletResponse`. Cookies are easy to understand and implement; however, there are drawbacks. If a Web site uses stored cookies to store sensitive information, operators of less reputable sites may read this information. There has been much controversy regarding privacy concerns related to how companies such as DoubleClick and Media Metrix use cookies to monitor user Web browsing and Web shopping patterns. For these reasons, some users disable cookies in their Web browser, not allowing cookies to be used as a session-tracking mechanism.

URL Rewriting allows you to track sessions by dynamically modifying the URL a user accesses to include additional information such as a session ID. However, this can result in frequent clashes with true paths and parameters that are appended to the URL. This technique is rarely used because there are easier and less intrusive methods of tracking sessions.

Hidden form fields allow session tracking by adding fields to an HTML form that aren't displayed by the client's form. The data stored in these hidden form fields are sent to the server when the form is submitted. A form field is concealed by declaring its type as hidden. For example, as a user shops, the product IDs of items a user has added to his shopping cart, can be added as hidden form fields. This technique is supported by all Web browsers and doesn't require any form of user authorization; however, its

usefulness is limited because it only works for a sequence of dynamically generated forms.

You can also use *stateful session Enterprise Java Beans* to capture the user state with an EJB implementation. This technique is often used with cookies for identifying the user. For example, a user's cookie could store a session ID created by a servlet, and the servlet could also store with a shopping cart session bean, the session ID it is associated with. To create a page that displays the user's shopping cart, a servlet could read the session ID from a cookie and use it as a key to locate the user's shopping cart via JNDI and home interface's finder methods.

HTTPSession is Sun's Java Servlet API class for user session tracking. A servlet engine will usually use a cookie to assign a user a temporary user ID, which is associated with an HTTPSession. The HTTPSession object then lives in the servlet engine until the user exceeds the default time limit for inactivity, the servlet engine shuts down, or session object is forcibly "invalidated" by a servlet. The HTTPSession object can be used to store and retrieve data that needs to be maintained for a session such as shopping cart information. Servlets can create per user and session objects and bind them to names (like JNDI) for use later in the same session. HTTPSession session tracking technique is similar to stateful session EJBs except HTTPSession manages data bearing objects in servlet container, whereas stateful session EJB manages data bearing objects in EJB container.

The Six Rules of Session Tracking

- ***Use cookies only for user identification.*** A user ID may be stored in the cookie so a user with can be greeted with the phrase "Welcome Back, Bob." Of course, "Bob" would still need to log in to perform more sensitive functions such as ordering, using his credit card number on file.
- ***Don't store any data besides a user's identification in cookies.*** This avoids privacy and security issues created by storing too much information in cookies. And never store sensitive data such as credit card numbers, social security numbers, phone numbers or addresses in a cookie.
- ***Use the HTTPSession tracking mechanism if a servlet (not EJB) implementation is used.*** And use stateful session EJBs when using an EJB implementation.
- ***Save any data that needs to be persistent for an indefinite time period to a backend database.*** A shopping application may provide a way to persist a shopping cart so a user can continue shopping another day. When the user accesses his shopping cart from the previous day, the data from the saved shopping card can be used to initialize a new shopping cart for the session.
- ***User authorization may be used for very simple applications.*** However, for most applications, this method of session tracking is insufficient.
- ***URL rewriting and hidden form field techniques should be avoided.*** The primary techniques used for session tracking should HTTPSession and stateful session EJBs in combination with a restricted use of cookies.

Since session tracking is critical for most non-trivial Web aplications, selecting the right session tracking mechanism is an important decision. Key factors to consider are security, ease of use and how well the technique melds with the overall system architecture. You should carefully examine the advantages and disadvantages of each method when building session tracking capability into a Web site.

RELATED ARTICLES

The Great Debate: PostgreSQL versus MySQL

Doing RIAs Right

Ingres Open Sources RAD Tool

ReadOnly Sessions and ASP.NET

Tunny, Colossus and Ada: Keeping an Open Mind

TOP 5 ARTICLES

No Top Articles.